WEST

Generate Collection Print

L5: Entry 46 of 66 File: USPT Mar 3, 1998

DOCUMENT-IDENTIFIER: US 5724589 A
TITLE: Development system with a property-method-event programming model for developing context-free reusable software components

Abstract Text (1):
A development system providing a property-method-event programming (PME) model for developing context-free reusable software components is described. Despite the absence of any C++ language support for events, the present invention provides a type-safe "wiring" mechanism--one using standard C++ to dispatch an event, raised by one object (the "event source"), to a method of another object (the "event sink"), with the requirement that the event source does not need to know the class of the event sink. As a result, the system allows developers to create C++ software components which can be connected together without the components having to know anything about the makeup of the component to which it is connected. Thus, developers can create pre-packaged, re-usable software components which can simply be "plugged into" a design--all accomplished within the confines of the standard C++ programming language (i.e., without having to employ proprietary extensions).

Brief Summary Text (9):
A "component" is a functional entity which can be completely characterized by its inputs and outputs. As a result, it can be used, and therefore tested, as a unit, independently of the context in which the component will eventually be used. It follows, therefore, that if a component can be characterized only by its inputs and outputs, then the details of its internal implementation will be completely hidden from the user. This aspect allows the implementation of a component to evolve over time without affecting its intended use.

Brief Summary Text (12):
Even with all of its object-oriented features, C++ really only helps one "create" software. It does little to help one with the use or re-use of that software. In a typical C++ object framework, the developer is provided with a programming model based on semifunctional "skeletons" or "base" classes, for the various "components" supported by the framework. The developer then needs to create "real" functioning components from these skeletons, usually by using the C++ inheritance mechanism. In effect, the designer starts by completing someone else's partial design. And to successfully complete such a process, the designer really needs to understand that design. The approach is characterized by "sub-classing" existing components, that is, adding real functionality to the "skeletons" which were provided by the framework. After completing this sub-classing process, the designer must still build and test the resulting "component" as part of his or her overall application. This is not very different from the cut-and-paste reuse of source code which has existed for some time. Simply put, present-day C++ frameworks do not provide fully functional, fully tested components which are ready for use without modification. A particular reason for this imitation is the fact that the C++ language really lets one create components with "inputs," in the form of C++ member functions, but there is no language-supported mechanism for "outputs." As a result, it is commonplace to create C++ components which make very specific assumptions about their surrounding environment. Such components become context bound, with severe restrictions on their use.

Brief Summary Text (13):
Consider a user interface control, such as a screen button, edit control, scroll bar, or the like. These serve as examples of components which, when implemented in typical C++ object frameworks, become context bound. A button object, for instance, would be implemented in a fashion which expressly assumes that it will also be

contained in another object which is derived from some class, say, a TControlEvents class. In typical implementation, these controls now communicate events to their containing object by "talking" directly to the "inputs" of this known TControIEvents object. Although the particular scheme varies from one C++ application framework to another, the end result is a set of controls which are context bound--they are constrained to communicate their events only through the methods defined by the containing object (here, the TControlEvents object). Clearly, new events cannot be added to existing controls. As a result, new controls with a different set of events cannot be used in the "old" container. For similar reasons, other containers--those which are not derived from the TControlEvents class--cannot be used at all.

Brief Summary Text (14):
Imagine the problems which would be encountered by an electrical engineer if a given electronic component could only be used as part of those assemblies whose "type" (i.e., detailed layout) were known at the time the component was created. For instance, what if a given electronic component could only be used on 3.times.5 inch circuit board, made of fiberglass-epoxy resin, positioned in the lower left corner of the circuit board, and it could only work correctly if the component to its left is a 4.7K resistor. Clearly today, such limitations would simply not be tolerated by electrical engineers. Instead, electronic components are designed to function in completely arbitrary electronic assemblies, as long as their inputs and outputs are "hooked up" correctly, that is, used in a context-free fashion.

Brief Summary Text (19):
"Closure" is the notion of binding or connecting "where to call" with "who is being called." Taken a step further, "closure," in accordance with the present invention, is the notion of binding a function pointer (i.e., where to call) with an object pointer (i.e., "who" is being called). Here, a closure may be thought of as a double pointer. It includes a pointer to an address to call together with a pointer to an address of an object to whom that address belongs. If a system has closure, one can specify a component completely in terms of its inputs and outputs. In particular, the user of that component need not know the class of the component to use it.

Brief Summary Text (20):
Given a language environment which does not have closure (e.g., C++), the issue remains how does one connect an "event source" to an "event sink." In the system of the present invention, this is accomplished by use of a dispatcher function which is given sufficient information so that it can "dispatch to" (i.e., call on to) the actual function (i.e., event sink) on the object, thereby achieving closure. In this manner, components of an arbitrary nature can be dealt with in the simple matter of "talking to" their inputs and "reacting to" their outputs, all without having to know anything about the makeup of a particular component. Moreover, the approach provides an early-bound, type-safe event model in standard C++.

Drawing Description Text (4):
FIG. 3 is a block diagram of a "black box" component, which may be characterized by its inputs and outputs.

Detailed Description Text (10):
Shown in further detail in FIG. 2, the development system 250 of the present invention includes a compiler 253, a linker 280, and an interface 255. Through the interface, the developer user supplies source modules 261 to the compiler 253. Interface 255 includes both command-line driven 259 and Integrated Development Environment (IDE) 257 interfaces, the former accepting user commands through command-line parameters, the latter providing menuing equivalents thereof. After tokenizing and parsing the source code or listings 261 and headers/includes files 251, the compiler 253 "compiles" or generates object module(s) 263. In turn, linker 280 "links" or combines the object modules 263 with libraries 271 to generate program(s) 265, which may be executed by a target processor (e.g., processor 101 of FIG. 1A). The standard libraries 271 include previously-compiled standard routines, such as graphics, I/O routines, startup code, math libraries and the like. The user developer may designate other libraries (e.g., custom libraries) whose code is to be linked into the target executable.

Detailed Description Text (16):
Consider a "black box" component, for instance, as shown in FIG. 3. Component 301 may be characterized by its inputs and outputs. In principle, a user of such a component need not know or be concerned about the "internals" which operate on the inputs to produce the outputs. In contrast, consider the C++ programming language,

with its class hierarchy mechanism. Objects created from C++ classes tend to expose an enormous amount of detail of the internal workings. C++ objects, in essence, can be viewed as "white box" objects, not "black box" objects. Inputs into a C++ object are the methods defined for the class from which the object is created. At the level of its methods, the internal workings of the object need not be surfaced to its client or user. In other words, the individual steps which comprise the various methods can be maintained in a black box fashion from users. Properties of the object, which are typically set using property-access methods of the class, can also be viewed in this light. Specifically, the getting and setting of properties can be achieved in a manner which is transparent to a user.

Detailed Description Text (17):
The difficulty remains, however, how one effectively handles the outputs of a component or object. Further, what exactly is an "output"? In the context of software development, an "output" is typically treated as an "event." Briefly stated, an event is something which has happened (i.e., an occurrence of an event) inside the component that a user of that component needs to react to (or at least be apprised of). It is these "outputs" which comprise the bulk of the problem of C++ components: no simple way exists in C++ of connecting an output of one component or object to the input of another. Thus, in C++, "closure" is not provided.

Detailed Description Text (20):
Taken a step further, "closure," in accordance with the present invention, is the notion of binding a function pointer (i.e., where to call) with an object pointer (i.e., "who" is being called). Here, a closure may be thought of as a double pointer. It includes a pointer to an address to call together with a pointer to an address of an object to whom that address belongs. If a system has closure, one can specify a component completely in terms of its inputs and outputs. In particular, the user of that component need not know the class of the component to use it. Since C++ does not provide closure, however, the problem remains as to how to "wire" the outputs in a convenient way.

Detailed Description Text (23):
Given a language environment which does not have closure (e.g., C++), the issue remains how does one connect an "event source" to an "event sink." In the system of the present invention, this is accomplished by use of a dispatcher function which is given sufficient information so that it can "dispatch to" (i.e., call on to) the actual function (i.e., event sink) on the object, thereby achieving closure. In this manner, components of an arbitrary nature can be dealt with in the simple matter of "talking to" their inputs and "reacting to" their outputs, all without having to know anything about the makeup of a particular component. Moreover, the approach provides an early-bound, type-safe event model in standard C++.

Detailed Description Text (39):
Context-fee reusable software components are implemented using a Property-Method-Event (PME) programming model of the present invention. In the PME model, "properties" and "methods" become the inputs to a component and "events" become the output from a component. Methods, which are supported directly by C++, impart behavior to a component in that they cause the component to perform some operation or action, such as asking an object or component to do something, including, for example, "move yourself," "paint yourself," or the like. Properties and events are not supported directly by C++. Accordingly, it is necessary to synthesize support for them, within the confines of standard C++.